

A Sparse Non-linear Classifier Design Using AUC Optimization*

Vishal Kakkar[†] Shirish K. Shevade[‡] S Sundararajan[§] Dinesh Garg[¶]

Abstract

AUC (Area under the ROC curve) is an important performance measure for applications where the data is highly imbalanced. Learning to maximize AUC performance is thus an important research problem. Using a max-margin based surrogate loss function, AUC optimization problem can be approximated as a pairwise rankSVM learning problem. Batch learning methods for solving the kernelized version of this problem suffer from scalability and may not result in sparse classifiers. Recent years have witnessed an increased interest in the development of online or single-pass online learning algorithms that design a classifier by maximizing the AUC performance. The AUC performance of nonlinear classifiers, designed using online methods, is not comparable with that of nonlinear classifiers designed using batch learning algorithms on many real-world datasets. Motivated by these observations, we design a scalable algorithm for maximizing AUC performance by greedily adding the required number of basis functions into the classifier model. The resulting sparse classifiers perform faster inference. Our experimental results show that the level of sparsity achievable can be order of magnitude smaller than the Kernel RankSVM model without affecting the AUC performance much.

1 Introduction.

In binary classification, a classifier is often trained by optimizing a performance measure such as accuracy. If the data is highly imbalanced, accuracy may not be a good measure to optimize. The all-positive or all-negative classifier may achieve good classification accuracy. But, this will result in misclassification of some important or rare events which typically belong to a minority class. Situations for which datasets are imbalanced are not uncommon in real-world applications and in such cases, classifiers are designed by optimizing measures other than accuracy [5].

Support Vector Machines (SVMs) have been very effective on several real-world problems. Standard

SVM formulations for binary classification problem assumes that misclassification costs are equal for both the classes. Therefore, SVMs are not suitable if the data is strongly imbalanced. Lin et. al. [16] proposed a simple extension of SVMs by using different penalization of positive and negative examples. This approach is useful if misclassification costs are known, which is typically not the case in practice. It is thus necessary to use a different measure for learning from imbalanced data.

AUC (Area Under ROC Curve) [17], [7] is an important performance measure and its optimization has been very effective, especially when class distributions are heavily skewed. However, computing the AUC is a costly operation as the AUC is written as a sum of pairwise losses between examples from different classes, which is quadratic in the number of training set examples. Further, the AUC is not a continuous function on the training set. This makes the optimization of AUC a challenging task.

Many algorithms have been designed to optimize AUC using surrogate loss functions (Herschtal et. al. [8], Joachims et. al. [9], Rudin et. al. [18], Kotlowski et. al. [13], Zhao et. al. [22]). Due to the high computational demands of the AUC or its variants, most of these algorithms are either one-pass algorithms or online algorithms which rely on sampling. Zhao et. al. [22] proposed an online AUC algorithm (OAM) which is based on the idea of reservoir sampling. This idea helps to represent all the received examples by the examples stored in buffers of fixed size. Gao et. al. [6] proposed a regression based algorithm for one-pass AUC (OPAUC) optimization. This algorithm maintains only the first and second order statistics of training data in memory, thereby resulting in a storage requirement which is independent of the training dataset size. Both these algorithms learn linear classifiers and are not directly suitable to design complex nonlinear decision boundaries, typically possible by using kernel classifiers.

Calders et. al. [3] proposed the use of polynomial approximations for the AUC, which can be computed in only one scan over the dataset. This approximation was used to design a linear classifier. Yang et. al. [21] proposed an online learning algorithm to optimize the AUC score by learning a nonlinear decision function via the kernel trick. This method, called online imbalanced

*Computer Science & Automation, IISc Bangalore, India.

†Computer Science & Automation, IISc Bangalore, India.

‡Microsoft Research, Bangalore, India.

¶IIT Gandhinagar, India.

learning with kernels (OILK), maintains a buffer to store the informative support vectors. Two buffer update policies, first-in-first-out and reservoir sampling were investigated. As the cost of determining the AUC score is very large, most of these algorithms avoid the exact computation of the AUC score and resort to online or one-pass approaches by making use of buffers to store the relevant information. Although the storage requirements are reduced for such methods, generalization performance of the resulting classifiers is not comparable with that of the nonlinear classifiers designed using batch learning algorithms on many real world datasets.

More relevant to the work in this paper is the large-scale Kernel RankSVM algorithm proposed by Kuo et. al. [14]. This algorithm, though designed for solving a ranking problem, can be extended to solve the AUC optimization problem. However, kernel evaluations are a bottleneck in training Kernel RankSVM. To alleviate this problem, it was proposed to store the full kernel matrix. Although this reduces repeated kernel evaluations, storage of the full kernel matrix is an issue if the dataset sizes are very large, as it requires $O(l^2)$ storage. Further, Kernel RankSVM may result in a model which uses a large number of support vectors, thereby incurring high inference cost.

Motivated by the above observations, we propose an algorithm to learn sparse models for maximizing AUC using a max-margin based surrogate loss function. Our experimental results show that the level of sparsity achievable can be order of magnitude smaller than the Kernel RankSVM model without affecting the AUC performance much. This helps to achieve significant speed-up during prediction. Due to the nature of our algorithm, parallelization is possible and we demonstrate that significant training speed-up is achievable by using a multi-core version of the algorithm.

Notation: Here we discuss the notations we have used in our work. All vectors will be column vector and row vectors will be denoted by a superscript, T . 2-norm of the vector \mathbf{x} is denoted by $\|\mathbf{x}\|$. $|J|$ denotes the cardinality of the set J . \mathbf{K} denotes the kernel matrix. $\mathbf{K}_{:,J}$ refers to the submatrix of \mathbf{K} made of all the l rows and the columns indexed by J . $\mathbf{K}_{i,J}$ refers to the i th row of the matrix \mathbf{K} . $\mathbf{K}_{J,J}$ refers to the submatrix of \mathbf{K} made of the rows indexed by J and the columns indexed by J .

2 Problem Definition

Let the training data be denoted by $\mathcal{D} = P \cup N$, where $P = \{\mathbf{x}_i^+, +1\}_{i=1}^p$, $N = \{\mathbf{x}_j^-, -1\}_{j=1}^n$ and $\mathbf{x}_i^+, \mathbf{x}_j^- \in R^d$. We will denote q^{th} training set example as \mathbf{x}_q . Let T denote the index set of pairs of positive and negative

instances in \mathcal{D} . Clearly, $|T| = pn$. Let $l = p + n$. Without loss of generality, we assume that $p \ll n$.

We assume that the non-linear decision function $f(\cdot)$ is an element of a Reproducing Kernel Hilbert Space (RKHS). That is, f is a linear combination of kernel functions,

$$(2.1) \quad f(\mathbf{x}) = \mathbf{w} \cdot \phi(\mathbf{x}) = \sum_{q=1}^l \beta_q \phi(\mathbf{x}_q) \cdot \phi(\mathbf{x}) = \sum_{q=1}^l \beta_q k(\mathbf{x}, \mathbf{x}_q),$$

where $\phi(\cdot)$ maps the data into high dimensions and $k(\cdot, \cdot)$ denotes a kernel function. The AUC score of the function f on the dataset \mathcal{D} is defined as

$$(2.2) \quad \begin{aligned} \text{AUC}(f) &= \frac{\sum_{i=1}^p \sum_{j=1}^n I(f(\mathbf{x}_i^+) > f(\mathbf{x}_j^-))}{pn} \\ &= 1 - \frac{\sum_{i=1}^p \sum_{j=1}^n I(f(\mathbf{x}_i^+) \leq f(\mathbf{x}_j^-))}{pn} \end{aligned}$$

where $I(\cdot)$ is the indicator function which outputs 1 if the argument is true and 0 otherwise. Thus maximizing $\text{AUC}(f)$ is equivalent to minimizing $\sum_{i=1}^p \sum_{j=1}^n I(f(\mathbf{x}_i^+) \leq f(\mathbf{x}_j^-))$. Writing $f(\mathbf{x}_i) = (\mathbf{K}\beta)_i$ and using a max-margin based surrogate loss function (a hinge or a squared hinge loss), we get the following two regularized formulations corresponding to the two loss functions:

$$(2.3) \quad \min_{\beta \in R^l} \frac{1}{2} \beta^T \mathbf{K} \beta + C \sum_{(i,j) \in T} \max(0, 1 - (\mathbf{K}\beta)_i + (\mathbf{K}\beta)_j)$$

and

$$(2.4) \quad \min_{\beta \in R^l} \frac{1}{2} \beta^T \mathbf{K} \beta + \frac{C}{2} \sum_{(i,j) \in T} \max(0, 1 - (\mathbf{K}\beta)_i + (\mathbf{K}\beta)_j)^2$$

where C is a positive hyperparameter that controls the error.

In this work, we focus on problem (2.4) as it is a continuously differentiable function and devise an efficient algorithm to solve it. Unlike typical classification problems where the loss function can be calculated for every single training set example, the second term in (2.4) involves losses defined over pairs of examples from different classes. This makes the problem (2.4) more challenging.

3 Related Work

We now briefly review some of the related works for AUC optimization.

Many online algorithms have been proposed to learn a linear classifier by maximizing the AUC score. These algorithms include Online AUC Maximization (OAM) [22] and Adaptive Online AUC Maximization (AdaOAM) [4]. AUC optimization in online learning is

a challenging task as the computation of the AUC score involves the sum of pairwise losses between instances from opposite classes. To tackle this challenge, online learning uses the idea of buffer sampling [22] [10]. A fixed size buffer is used to represent all the observed data by storing some randomly sampled examples in it. Kar et. al. [10] introduced the idea of stream subsampling with replacement as the buffer update strategy. Although these online algorithms have demonstrated good AUC performance by using simple online gradient descent approaches, they do not use the geometrical knowledge of the observed data. AdaOAM overcomes this limitation by employing an adaptive gradient method that exploits the knowledge of historical gradients. Its variant, SAdaOAM was proposed to design a sparse model in online AUC maximization task. Gao et. al. [6] proposed a one-pass optimization algorithm by considering square loss for the AUC optimization task. Due to the use of squared error loss, the algorithm only needs to store the first and second order statistics for the observed data.

A main drawback of the online methods discussed above is that they learn a linear classifier and do not exploit the learning power of kernel methods. To address this issue, yang et. al. [21] investigated Online Imbalanced Learning with Kernels (OILK) where informative support vectors are stored in the buffer. Two buffer update strategies, First-In-First-Out (FIFO) and Reservoir Sampling (RS) were investigated. By conducting experiments on real-world datasets, it was demonstrated that the kernel methods for AUC maximization performed better than their linear classifier counterparts. The proposed method [21] is however an online algorithm.

Joachims [9] presented a structural SVM framework for optimizing AUC in a batch mode. By formulating (2.4) as a 1-slack structural SVM problem, Joachims [9] solved its dual problem by a cutting plane method. The method, though initially designed for linear classifiers, can be easily extended to nonlinear classifiers. Numerical experiments showed that, for ranking learning problems, this method is slower than others state-of-the-art methods that solve (2.4) directly [14] [15].

Learning to rank is an important supervised learning problem and has application in a variety of domains such as information retrieval and online advertising. Treating the all instances query number same, the set of preference pairs will be same as T and the Kernel rankSVM problem, discussed in [14] is same as (2.4). Kuo et. al. [14] used trust region Newton method to solve this problem. This method stores the full kernel matrix as repeated kernel evaluations are bottleneck in Kernel rankSVM. This method has two drawbacks:

1) it is not scalable as the memory requirement is prohibitively high for large datasets, and 2) the learned model is not sparse resulting in computationally expensive predictions.

4 Our Approach: Sparse Kernel AUC

Our aim is to learn a sparse nonlinear classifier model for a binary classification problem with imbalanced data distributions for the two classes. We now discuss our approach to solve (2.4). A similar problem formulation was used in [14] to solve the problem of learning to rank and the algorithm designed there is also applicable to our setting. Kuo et. al. [14] alleviated the difficulty of computing the loss term, which involves summation over preference pairs, by using order-statistic trees. Although the cost of computing the required quantities was reduced to $O(l \log l)$ from $O(l^2)$, the kernel evaluations amount to $O(ld)$ time, which can be reduced to $O(l)$ if the kernel matrix \mathbf{K} is maintained throughout the optimization algorithm. In their implementation, Kuo et. al. [14] store the full kernel matrix \mathbf{K} which is a dense matrix of size $l \times l$. However, for large datasets it is impractical to store the full kernel matrix \mathbf{K} in the main memory. Further, for such huge datasets, the resulting classifier may not be sparse, thereby making the inference slow. It is therefore desired to devise a different approach to solve (2.4) and design a sparse classifier.

Motivated by the success of the matching pursuit approach, presented by Keerthi et. al. [11], to design sparse SVM classifiers, we propose a new and efficient algorithm to solve (2.4) using matching pursuit ideas. The algorithm requires to compute and maintain the kernel matrix of size $l \times d_{max}$ (where d_{max} is the user specified positive parameter whose value can be about 5 – 10% of the dataset size l) which helps to reduce the memory requirement considerably. For the dataset with $l = 49,990$, we observed that $d_{max} \approx 200$ was sufficient to achieve very good AUC performance on the test set.

We also demonstrate that efficient computations of the objective function in (2.4), gradient and Hessian-vector product computations are done by using simple techniques like sorting, binary search and hashing [12] and do not require the use of sophisticated data structures such as order-statistic trees. As our experimental results show, the proposed approach is faster than the approach of Kuo et. al. [14] applied to the AUC maximization problem and achieves comparable generalization performance using small number of support vectors.

We now discuss the key components of our proposed algorithm.

4.1 Reformulation Borrowing the ideas presented in [11], we maintain a set of greedily chosen kernel basis functions to design a sparse non-linear classifier. The cardinality of this set is denoted by d_{max} , a user specified positive integer. Let J denote the index set of these basis functions. In our experiments, we choose $J \subset \{1, 2, \dots, l\}$. Having defined the set J , the parameter vector \mathbf{w} in (2.1) can be represented as

$$\mathbf{w} = \sum_{q \in J} \beta_q \phi(\mathbf{x}_q)$$

and the problem formulation in (4) can be written as (4.5)

$$\min_{\beta_J \in R^{|J|}} \mathbf{E}(\beta_J) \equiv \frac{1}{2} \beta_J^T \mathbf{K}_{J,J} \beta_J + \frac{C}{2} \sum_{(i,j) \in T} \max(0, 1 - \mathbf{K}_{i,J} \beta_J + \mathbf{K}_{j,J} \beta_J)$$

Note that the Kernel rankSVM algorithm solves the following problem;

$$(4.6) \quad \min_{\beta \in R^l} \frac{1}{2} \beta^T \mathbf{K} \beta + \frac{C}{2} \sum_{(i,j) \in S} \max(0, 1 - (\mathbf{K} \beta)_i + (\mathbf{K} \beta)_j)^2$$

where $S = \{(i, j) | q_i = q_j, y_i > y_j\}$ is the set of preference pairs for queries q . This problem requires either to store the full kernel matrix \mathbf{K} or requires many kernel evaluations, which become a bottleneck. On the other hand, the solution to our problem (4.5) requires to store the matrix of size $l \times d_{max}$, which makes it scalable.

In this work, we solve (4.5) using matching pursuit ideas [20], [11]. In this approach, starting with $J = \emptyset$, a training set example is chosen from the set $\{1, 2, \dots, l\} \setminus J$ such that its inclusion in the set J results in a maximum improvement in the objective function. The optimization problem is then solved with respect to β_J . This procedure is repeated till $|J| = d_{max}$ holds true. Algorithm 1 gives the pseudo-code of this procedure. Step 5 of this algorithm is computationally expensive and in section V, we will discuss some approaches to make it efficient

The efficiency of this algorithm depends on the efficient computation of the objective function value $\mathbf{E}(\beta_J)$, its gradient $\nabla \mathbf{E}(\beta_J)$ and Hessian-vector product $\nabla^2 \mathbf{E}(\beta_J) \mathbf{v}$ for any vector $R^{|J|}$. If \mathbf{A} a pairwise indexing matrix and \mathbf{A}_{β_J} denotes the indexing matrix of violating pairs, which contribute to the loss function, then by defining

$$(4.7) \quad \mathbf{u}_{\beta_J} = \mathbf{A}_{\beta_J}^T \mathbf{A}_{\beta_J} \mathbf{K}_{\cdot,J} \beta_J,$$

the problem in (4.5) can be re-written as (4.8)

$$\min_{\beta_J} \mathbf{E}(\beta_J) \equiv \frac{1}{2} \beta_J^T \mathbf{K}_{J,J} \beta_J + \frac{C}{2} (\beta_J^T \mathbf{K}_{\cdot,J}^T (\mathbf{u}_{\beta_J} - 2 \mathbf{A}_{\beta_J}^T \mathbf{e}_{\beta_J}) + p_{\beta_J}). \quad \text{---} \quad \text{1} d_{max} \text{ is for computing } \mathbf{K}_{\cdot,J} \text{ and } l \log l \text{ is for sorting}$$

where p_{β_J} is the number of violating pairs (details given in appendix). This rewriting helps in computing $\mathbf{E}(\beta_J)$, $\nabla \mathbf{E}(\beta_J)$ and $\nabla^2 \mathbf{E}(\beta_J) \mathbf{v}$ efficiently as all of these quantities require the computation of \mathbf{u}_{β_J} . By defining

$$(4.9) \quad SV(\beta_J) = \{(i, j) \in T \mid 1 - \mathbf{K}_{i,J} \beta_J + \mathbf{K}_{j,J} \beta_J > 0\}$$

and

$$\begin{aligned} SV_i^+(\beta_J) &\equiv \{j \mid (j, i) \in SV(\beta_J)\}, \quad l_i^+(\beta_J) \equiv |SV_i^+(\beta_J)|, \\ \gamma_i^+(\beta_J, \mathbf{v}) &\equiv \sum_{j \in SV_i^+(\beta_J)} \mathbf{K}_{j,J}^T \mathbf{v}, \\ SV_i^-(\beta_J) &\equiv \{j \mid (i, j) \in SV(\beta_J)\}, \quad l_i^-(\beta_J) \equiv |SV_i^-(\beta_J)|, \\ \gamma_i^-(\beta_J, \mathbf{v}) &\equiv \sum_{j \in SV_i^-(\beta_J)} \mathbf{K}_{j,J}^T \mathbf{v}. \end{aligned}$$

one compute \mathbf{u}_{β_J} efficiently, as can be seen from equation (??) in appendix.

Algorithm 1: Sparse Classifier Design Algorithm

Input: $\mathcal{D} = \{\mathbf{x}_i^+, +1\}_{i=1}^p \cup \{\mathbf{x}_j^-, -1\}_{j=1}^n$, C , d_{max}

Output: J , β_J

- 1: $J = \emptyset$
 - 2: **while** $|J| < d_{max}$ **do**
 - 3: select a new basis function j^* which gives a maximum decrease in the objective function \mathbf{E}_{β_J}
 - 4: $J = J \cup \{j^*\}$
 - 5: Optimize the objective function w.r.t β_J
 - 6: **end while**
-

Lee et. al. [15] and Airola et. al. [1] used order-statistic trees to efficiently compute the $l_i^+(\beta_J)$ and $l_i^-(\beta_J)$ for rankSVM. The problem of maximizing AUC does not require order-statistic trees. It is enough to use sorting, searching and hashing methods. The details are given in Algorithm 2.

For a given β_J , we define the set of ordered pairs which contributes to the empirical loss of objective function in (4.5) as $SV(\beta_J)$. For every example in the training set, by finding out the set of violating examples of the other class (SV^+ and SV^-) and the sum of $\mathbf{K}_{\cdot,J}$ for those examples (γ^+ and γ^-), we can compute the empirical loss term in (4.5). These computations can be done efficiently by using sorting (Steps 1-7), hashing (Steps 9-12) and searching (Steps 15-26). The complexity of this algorithm is $O((d_{max} + \log l)^1)$, which is better than naive computation of pairwise losses in (4.5). Further, in our experiments we implemented steps 15-26 of Algorithm 2 in multi-core setting. Empirical evaluation, discussed in the next section, shows that this resulted in a significant speed up of our algorithm.

Algorithm 2: Calculating $l_i^+(\beta_J)$, $l_i^-(\beta_J)$, $\gamma_i^+(\beta_J, \mathbf{v})$, and $\gamma_i^-(\beta_J, \mathbf{v})$

Input: $\mathbf{K}_{:,J}$, β_J , \mathbf{v} , P , N

Output: $l_i^+(\beta_J)$, $l_i^-(\beta_J)$, $\gamma_i^+(\beta_J, \mathbf{v})$ and $\gamma_i^-(\beta_J, \mathbf{v})$

```

1: scoreP = zeros(2, |P|), scoreN = zeros(2, |N|)
2: scoreP[1] =  $\mathbf{K}_{i,J} * \beta_J$ , for all  $i \in P$ 
3: scoreP[2] =  $\mathbf{K}_{i,J} * \mathbf{v}$ , for all  $i \in P$ 
4: sort scoreP w.r.t to first row
5: scoreN[1] =  $\mathbf{K}_{j,J} * \beta_J$ , for all  $j \in N$ 
6: scoreN[2] =  $\mathbf{K}_{j,J} * \mathbf{v}$ , for all  $j \in N$ 
7: sort scoreN w.r.t to first row
8: scorePsum = scoreP[2], scoreNsum = scoreN[2]
9: for  $i = 2$  to  $|P|$  do
10:   scorePsum[i] = scorePsum[i] + scorePsum[i-1]
11: end for
12: for  $i = |N| - 1$  to  $1$  do
13:   scoreNsum[i] = scoreNsum[i] + scoreNsum[i+1]
14: end for
15: for  $i = 1$  to  $|P|$  do
16:   score =  $(\mathbf{K}_{i,J} * \beta_J) - 1$ 
17:   find the index k of scoreN using binary search
     s.t.  $scoreN[k-1] < score \leq scoreN[k]$ 
18:    $l_i^-(\beta_J) = length(k : |N|)$ 
19:    $\gamma_i^-(\beta_J, \mathbf{v}) = scoreNsum[k]$ 
20: end for
21: for  $j = 1$  to  $|N|$  do
22:   score =  $(\mathbf{K}_{j,J} * \beta_J) + 1$ 
23:   find the index k of scoreP using binary search
     s.t.  $scoreP[k] \leq score < scoreP[k+1]$ 
24:    $l_j^+(\beta_J) = k$ 
25:    $\gamma_j^+(\beta_J, \mathbf{v}) = scorePsum[k]$ 
26: end for

```

4.2 Basis Selection: We now discuss how to choose the kernel basis functions for a given problem. Our approach is greedy [11] and starts with an empty set J . A training set example is chosen from the set $\{1, 2, \dots, l\} \setminus J$ such that its inclusion in the set J results in a maximum improvement in the objective function. The optimization problem is then solved with respect to β_J . This procedure is repeated till $|J| = d_{max}$ holds true. Algorithm 1 gives the details of this procedure. The efficiency of this procedure depends on the optimization method used to solve (5). We discuss the two methods to add basis functions in the set J .

4.2.1 Method 1 A straightforward method is to choose every $q \in \mathcal{D} \setminus J$ and include in J (i.e., $J = J \cup q$), optimize (4.5) fully using (β_J, β_q) and calculate the improvement in the objective function. Let it be \mathbf{E}_q .

Choose

$$j^* = \arg \min_{q \in \mathcal{D} \setminus J} \mathbf{E}_q$$

in Step 3 of Algorithm (1). But solving (4.5) fully, $|\mathcal{D} \setminus J| \approx O(l)$ times is computationally expensive. Instead of choosing every $q \in \mathcal{D} \setminus J$, we can work with a smaller subset of size κ in $\mathcal{D} \setminus J$. Smola in [19] suggested this random subset choice (of size 59), for Gaussian process regression successfully. But even with κ number of random examples, this method of selection is still quite computationally heavy.

4.2.2 Method 2 In method 1, we solve a $|J| + 1$ dimensional problem, optimizing (β_J, β_q) to solve (4.5) completely. Instead it may be good idea to fix β_J and solve (4.5) for β_q , to determine \mathbf{E}_q . This problem is easy to solve as it is a one-dimensional problem.

$$\begin{aligned}
& \min_{\beta_q} \frac{1}{2} \begin{pmatrix} \beta_J^T & \beta_q \end{pmatrix} \begin{pmatrix} \mathbf{K}_{J,J} & \mathbf{K}_{J,q} \\ \mathbf{K}_{q,J} & \mathbf{K}_{q,q} \end{pmatrix} \begin{pmatrix} \beta_J \\ \beta_q \end{pmatrix} + \\
(4.10) \quad & \frac{C}{2} \sum_{(i,j) \in T} \max(0, 1 - (\mathbf{K}_{i,J} - \mathbf{K}_{i,j})\beta_J - (\mathbf{K}_{i,q} - \mathbf{K}_{j,q})\beta_q)^2
\end{aligned}$$

Keerthi et. al. [11] showed to solve this one dimensional problem in $O(l)$ time for SVM by using Newton-Raphson type iterations. So in our case complexity of solving this one dimensional problem will be $O(l(d_{max} + \log l))$. Here also instead of choosing $q \in \mathcal{D} \setminus J$, we choose q in random sample of size κ .

4.3 Truncated Newton Optimization Method

The function $\mathbf{E}(\beta_J)$ (4.5) which we consider, can be optimized using second order optimization method. We use Truncated Newton Optimization Method (TRON) instead of Classical Newton Method, because in the classical newton method the update step is $\beta = \beta - H^{-1}g$. Computation of the Hessian and its inverse is a computational intensive task. Therefore, to reduce the computation time, we uses Truncated Newton Iteration to optimize current β_J . We use the linear conjugate gradient iteration in this method to approximate the $H^{-1}g$ which uses the Hessian-vector product for some vector \mathbf{v} . As we discussed in the Section IV that, we can compute Hessian-vector product efficiently with overall complexity

$$O(l(d_{max} + \log l))$$

We have not discuss the details of the linear conjugate gradient iteration. The details of the steps involved in linear conjugate gradient algorithm can be found in [2]. There are many variations around it, all of them rely on Hessian vector multiplications. In our

Algorithm 3: Truncated Newton Iteration

Input: J , C , current β_J

Output: Optimized β_J

- 1: $\beta_J^0 = \beta_J$, $k = 0$
 - 2: **while** β_J^k is not optimal for obj fun **do**
 - 3: Get direction d using linear CG (which requires Hessian-vector product & gradient) method
 - 4: Find step size t using line search
 - 5: Update $\beta_J^{k+1} = \beta_J^k + td$
 - 6: Set $k=k+1$
 - 7: **end while**
-

implementation we use the **minres** function from MATLAB to get the direction by using the Hessian-vector product. The pseudo code for Truncated Newton iteration to solve (??) is given in the Algorithm 3.

4.4 Computational Complexity Assuming that kernel matrix \mathbf{K} is stored in memory, then the computation of the loss term in (4.5) will require $O(l(d_{max} + \log l))$ computation time. On the other hand the corresponding term in (4.6) require the computation time of $O(l^2)$. For large datasets it may not be feasible to store \mathbf{K} in main memory. Therefore, for such datasets, Kernel rankSVM resorts to several block wise computations of \mathbf{K} , which may result in increased training time. This problem does not arise in our approach, as the maximum sub-matrix of \mathbf{K} that it needs to store is of size $l \times d_{max}$.

5 Empirical Evaluation

In this section, we discuss the experimental evaluations of the proposed algorithm for sparse classifier design. In particular, we demonstrate that the proposed Sparse Kernel AUC algorithm results in a sparser classifier and gives comparable generalization performance with the Kernel rankSVM algorithm. Further, batch learning algorithms perform better than online learning algorithms on majority of real world datasets.

In our experiments, we used Gaussian kernel function, $K(x_i, x_j) = \exp(-\frac{1}{2\sigma^2}\|x_i - x_j\|^2)$ where $\sigma > 0$, for all the experiments. The kernel parameter σ and regularization hyper-parameter C were tuned using cross-validation. For this, a grid of (C, σ) values, where $C \in \{10^{-5}, 10^{-4}, \dots, 10^5\}$ and $\sigma \in \{2^{-5}, 2^{-4}, \dots, 2^5\}$ was searched. AUC performance corresponding to the (C, σ) pair, which gave the best validation set AUC performance, is reported. The value d_{max} was set to $\frac{l}{2}$. The proposed algorithm was terminated when $|J| = d_{max}$ was true or there was not significant change in the validation set AUC performance. All the experiments were

performed using MATLAB implementations on a Intel(R) Xeon(R) CPU E5620@2.40GHz machine with 16 cores and 16 GB main memory under Linux.

We compare the following methods: 1) Sparse Kernel AUC: our proposed sparse AUC optimization approach discuss in Section IV. 2) Kernel rankSVM: an extension of Kernel rankSVM method, discussed in [14], to the AUC optimization problem. 3) Online Imbalanced Learning with Kernels (OILK) [21], and 4) Adaptive Gradient Method for Online AUC Maximization (AdaOAM) [4]. The performance of these methods was compared in terms of the AUC score on the test set (if a test set is available). If the test set is not explicitly available, AUC score on validation set, averaged over 4 independent run of five-fold splits of each dataset is reported. Since the aim of this paper is to design non-linear sparse classifier model using AUC optimization, we report the number of support vectors present in the final model for batch learning methods: Sparse Kernel AUC and Kernel rankSVM. The other two methods use an online learning approach and it may not be fair to compare the number of support vectors obtained using them with those obtained using batch learning methods. CPU time comparison of batch learning methods: Sparse Kernel AUC and Kernel rankSVM was not done as the implementations were done using MATLAB and C programming language respectively. But we compare computational complexity of both methods in Section IV-D.

We used 14 benchmark datasets to compare our proposed method, Sparse Kernel AUC, with other three methods. The dataset details are given in Table I. The datasets are available at UCI² or LIBSVM³ dataset repository. Some multi-class datasets (glass, vechile, poker) were converted to class imbalanced binary datasets. As given in Table I, training+test splits are not available for some datasets.

Effect of retraining and κ To make Algorithm 1 efficient, it may be a good idea to perform conjugate gradient optimization in step 5 only from time to time. We experimented with 3 retraining strategies where step 5 is executed after the addition of 1) every basis function (i.e always), 2) $|J| = \lfloor 2^{0.25} \rfloor$ basis functions and 3) $|J| = 2^j$, $j = 0, \dots$, basis functions. The results are presented in figure (1). It is clear from this figure that, always retraining increases the training time. Similar generalization performance is achieved in other cases of retraining. We found that $\lfloor 2^{0.25} \rfloor$ was a good choice across many datasets and used it in our experiments.

As mentioned in section IV-B, instead of choosing a

²<https://archive.ics.uci.edu/ml/datasets.html>

³<https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/>

Datasets	Sparse Kernel AUC	Kernel rankSVM	$OILK_{RS}$	AdaOAM
sonar	0.914 ± 0.043 (105)	0.951 ± 0.029 (167)	$0.929 \pm .039$	-
glass	0.871 ± 0.054 (150)	0.881 ± 0.051 (171)	-	0.816 ± 0.058
ionosphere	0.980 ± 0.017 (182)	0.987 ± 0.014 (281)	0.954 ± 0.021	-
balance	1.000 ± 0.000 (6)	1.000 ± 0.000 (500)	-	0.579 ± 0.106
australian	0.913 ± 0.034 (256)	0.930 ± 0.020 (552)	0.925 ± 0.021	0.927 ± 0.016
vechile	0.977 ± 0.022 (431)	0.995 ± 0.002 (677)	-	0.818 ± 0.026
fourclass	0.999 ± 0.000 (108)	1.000 ± 0.000 (690)	0.829 ± 0.036	-
svmguid3	0.823 ± 0.027 (216)	0.824 ± 0.026 (995)	-	0.734 ± 0.038
a2a	0.880 ± 0.009 (64)	0.880 ± 0.010 (1741)	-	0.873 ± 0.019
magic04	0.874 ± 0.023 (182)	0.894 ± 0.007 (15124)	-	0.798 ± 0.007

Table 2: Validation set AUC Performance (mean \pm std.) and maximum number of basis functions (in parenthesis) comparison of various methods. The AUC performance numbers for $OILK_{RS}$ and AdaOAM are reported from [4] and [21] respectively.

Datasets	Sparse Kernel AUC	Kernel rankSVM	$OILK_{RS}$	AdaOAM
segment	0.996 (91)	0.998 (210)	0.997 ± 0.003	-
satimage	0.961 (431)	0.969 (4435)	0.896 ± 0.024	-
ijcnn1	0.995 (182)	1.000 (49990)	-	-
poker	0.668 (363)	0.674 (25010)	-	0.571 ± 0.007

Table 3: Test set AUC Performance and number of basis functions (in parenthesis) comparison of various methods. The AUC performance numbers for $OILK_{RS}$ and AdaOAM are reported from [4] and [21] respectively.

Datasets	#train inst	#test inst	#dim	n/p
sonar	208	-	60	1.144
glass	214	-	9	2.057
ionosphere	351	-	34	1.785
balance	625	-	4	11.755
australian	690	-	14	1.247
vechile	846	-	18	3.251
fourclass	862	-	2	1.807
svmguid3	1,243	-	22	3.199
a2a	2,265	-	123	2.959
magic04	19,020	-	10	1.843
segment	210	2,100	19	6.000
satimage	4,435	2,000	36	9.279
ijcnn1	49,990	91,701	22	10.0
poker	25,010	1,000,000	11	20.0

Table 1: Details of datasets

possible basis function from $\mathcal{D} \setminus J$, we chose a subset κ of examples from this set as possible candidate for basis functions. Different values of κ (1, 10, 100) were tried. The results are shown in figure (2). Although these 3 values of κ resulted in similar steady state generalization performance, it was observed that for $\kappa = 100$ steady state generalization performance was achieved faster.

So, $\kappa = 100$ was a good choice.

Discussion From tables II and III, we observed that the generalization performance of the proposed Sparse Kernel AUC method is comparable with that of the Kernel rankSVM method. Both these batch learning methods perform significantly better than the $OILK$ method on ionosphere, fourclass and satimage datasets. The kernel based methods, Sparse Kernel AUC, Kernel rankSVM and $OILK_{RS}$ perform better than linear classifier based method (AdaOAM) on majority of the datasets.

The proposed method required smaller number of basis functions than those required by Kernel rankSVM to achieve comparable AUC performance. Thus the proposed method is recommended for designing sparse classifiers for large datasets.

The reduction in the number of basis vectors is two orders of magnitude in case of some large datasets (magic, poker and ijcnn1).

Experiments in Multi-core Setting To study the speed-up of our proposed algorithm in multi-core environment, we parallelized steps 15-26 of Algorithm 2. The speed-up was studied on three large datasets by gradually increasing the number of cores from 1 to 16. Figure (3) depicts the time comparison. It is clear from

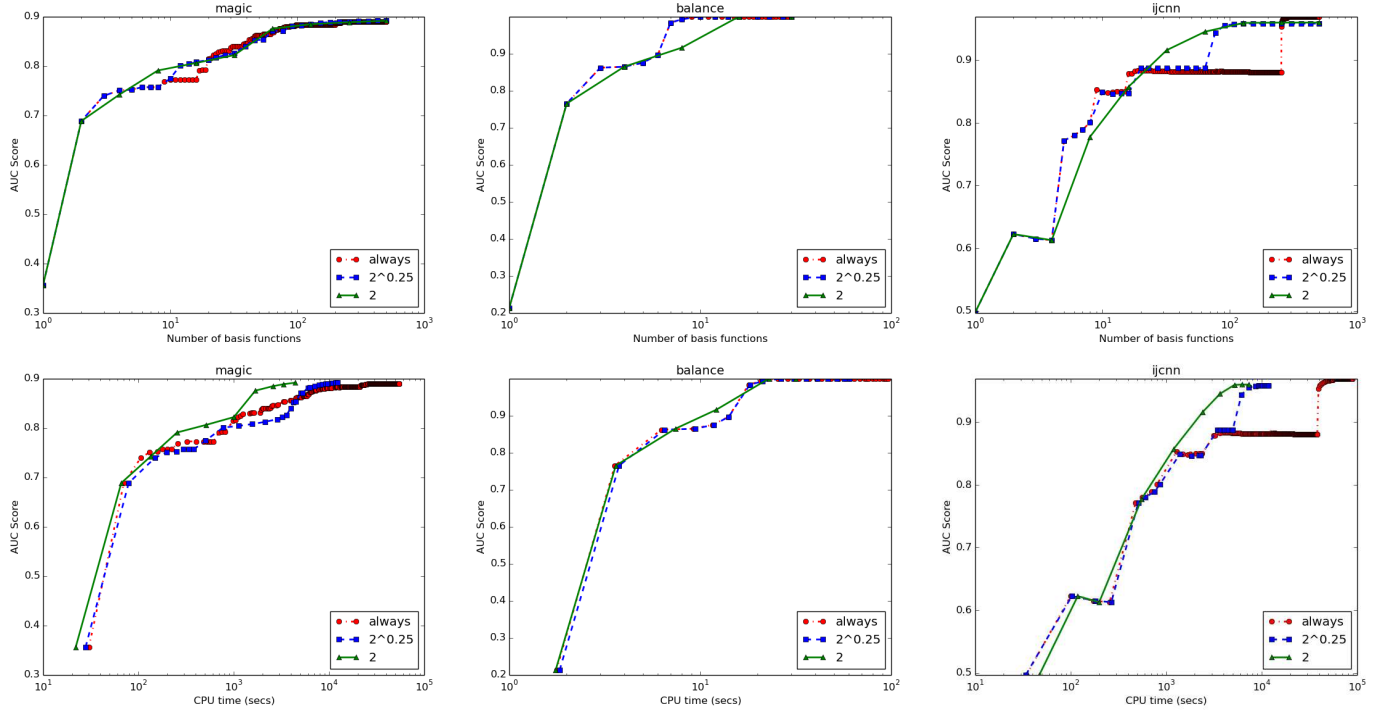


Figure 1: Three different retraining strategies showing a different trade-off between AUC and time, always retraining is too time consuming.

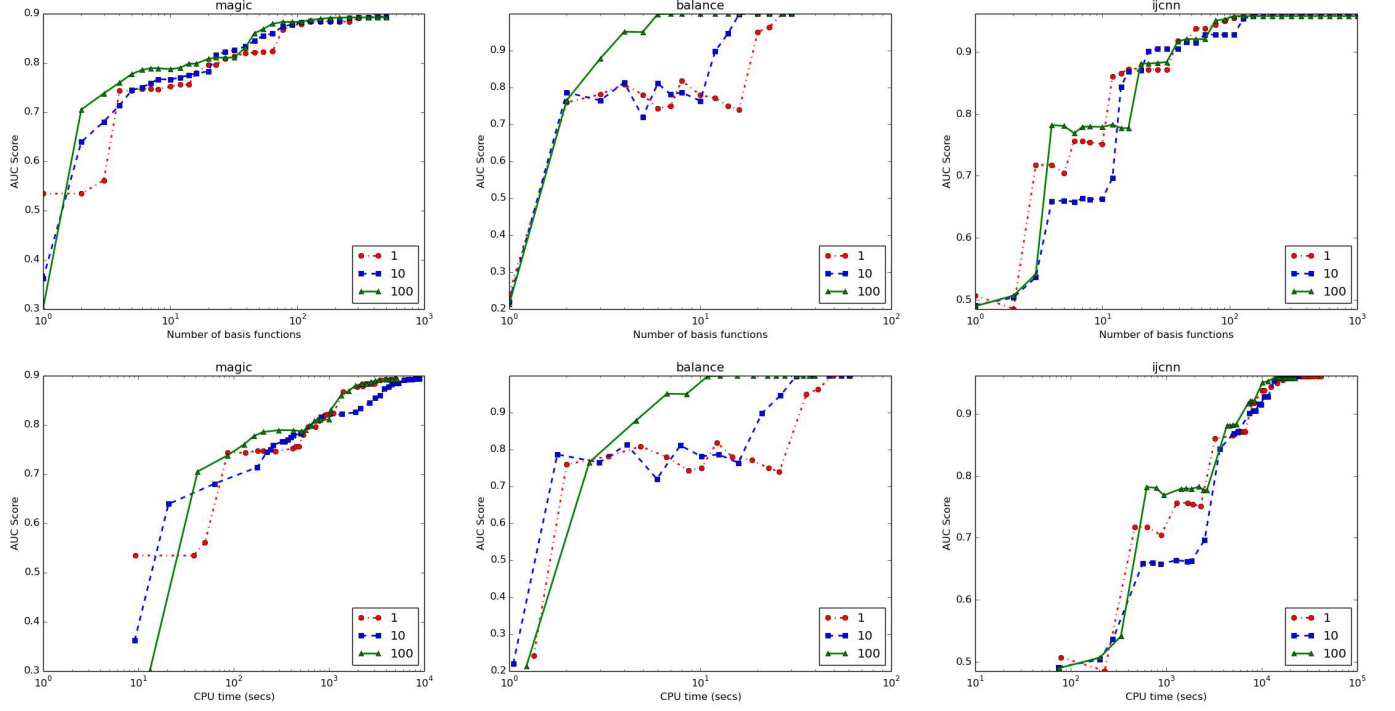


Figure 2: Influence of the parameter κ : performance is not much affected, but the computational cost is a bit larger when $\kappa=1$. $\kappa = 100$ seems a good trade-off.

this figure that significant speed-up can be obtained by running our method in multi-core environment. The speed-up is noticeably on large datasets like ijcnn1. Detailed investigation is however needed to study the parallelization of the complete proposed algorithm.

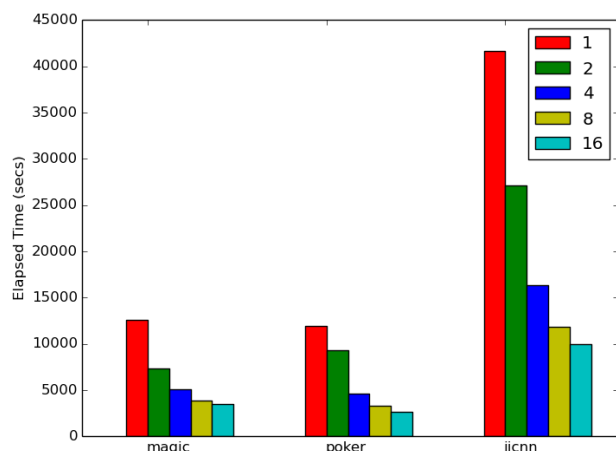


Figure 3: Effect of increasing the numbers of core on time is shown for 3 benchmark datasets.

6 Conclusion

This paper studied a new and efficient learning algorithm to design a sparse nonlinear classifier using AUC maximization. The algorithm tackles the challenge of lengthy training times of kernel methods by greedily adding the required number of basis functions in the model. We demonstrated that the resulting sparse classifier achieved comparable generalization performance with that achieved by full models. On many large datasets, it was observed that the proposed algorithm results in using significantly small number of basis functions in the model. We also demonstrated that batch learning algorithms for AUC optimization perform better than online algorithms on many datasets. We are currently investigating the extension of these ideas to a distributed setting. The MATLAB code for this paper is available at this dropbox link https://www.dropbox.com/s/ha7w3o0291hb5bn/ICDM_AUCCode.tar.gz?dl=0

References

- [1] Antti Airola, Tapio Pahikkala, and Tapio Salakoski. Training linear ranking svms in linearithmic time using red-black trees. *Pattern Recognition Letters*, 32(9):1328–1336, 2011.
- [2] Richard Barrett, Michael W Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk

- Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. Siam, 1994.
- [3] Toon Calders and Szymon Jaroszewicz. Efficient auc optimization for classification. In *Knowledge Discovery in Databases: PKDD 2007*, pages 42–53. Springer, 2007.
- [4] Yi Ding, Peilin Zhao, Steven CH Hoi, and Yew-Soon Ong. Adaptive subgradient methods for online auc maximization. *arXiv preprint arXiv:1602.00351*, 2016.
- [5] Tom Fawcett. Using rule sets to maximize roc performance. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 131–138. IEEE, 2001.
- [6] Wei Gao, Rong Jin, Shenghuo Zhu, and Zhi-Hua Zhou. One-pass auc optimization. *arXiv preprint arXiv:1305.1363*, 2013.
- [7] James A Hanley and Barbara J McNeil. A method of comparing the areas under receiver operating characteristic curves derived from the same cases. *Radiology*, 148(3):839–843, 1983.
- [8] Alan Herschtal and Bhavani Raskutti. Optimising area under the roc curve using gradient descent. In *Proceedings of the twenty-first international conference on Machine learning*, page 49. ACM, 2004.
- [9] Thorsten Joachims. A support vector method for multivariate performance measures. In *Proceedings of the 22nd international conference on Machine learning*, pages 377–384. ACM, 2005.
- [10] Purushottam Kar, Bharath K Sriperumbudur, Prateek Jain, and Harish C Karnick. On the generalization ability of online learning algorithms for pairwise loss functions. *arXiv preprint arXiv:1305.2505*, 2013.
- [11] S Sathiya Keerthi, Olivier Chapelle, and Dennis DeCoste. Building support vector machines with reduced classifier complexity. *The Journal of Machine Learning Research*, 7:1493–1515, 2006.
- [12] Donald E Knuth. The art of computer programming, vol. 3: Sorting and searching,” 1973.
- [13] Wojciech Kotlowski, Krzysztof J Dembczynski, and Eyke Huellermeier. Bipartite ranking through minimization of univariate loss. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1113–1120, 2011.
- [14] Tzu-Ming Kuo, Ching-Pei Lee, and Chih-Jen Lin. Large-scale kernel ranksvm. In *SDM*, pages 812–820. SIAM, 2014.
- [15] Ching-Pei Lee and Chuan-bi Lin. Large-scale linear ranksvm. *Neural computation*, 26(4):781–817, 2014.
- [16] Yi Lin, Yoonkyung Lee, and Grace Wahba. Support vector machines for classification in nonstandard situations. *Machine learning*, 46(1-3):191–202, 2002.
- [17] Charles E Metz. Basic principles of roc analysis. In *Seminars in nuclear medicine*, volume 8, pages 283–298. Elsevier, 1978.
- [18] Cynthia Rudin and Robert E Schapire. Margin-based ranking and an equivalence between adaboost and rankboost. *The Journal of Machine Learning*

Research, 10:2193–2232, 2009.

- [19] Alex J Smola and Peter Bartlett. Sparse greedy gaussian process regression. In *Advances in Neural Information Processing Systems 13*. Citeseer, 2001.
- [20] Pascal Vincent and Yoshua Bengio. Kernel matching pursuit. *Machine Learning*, 48(1-3):165–187, 2002.
- [21] Haiqin Yang, Junjie Hu, Michael R Lyu, and Irwin King. Online imbalanced learning with kernels. In *NIPS Workshop on Big Learning*, 2013.
- [22] Peilin Zhao, Rong Jin, Tianbao Yang, and Steven C Hoi. Online auc maximization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 233–240, 2011.